Building and using static and shared ‘c’ Libraries

One of the problems with developed programs, is that they tend to grow larger and larger, bringing up overall compilation and linking time to a large figure, and polluting out makefile, and the directory where we placed the source files. The first time a program we write reaches this state, is normally when we look for a different way to manage our projects.

It is this point where we start thinking about combining out source code into small units of related files, that can be managed with a separate makefile, possibly by a different programmer (for a multi-programmer project).

A library is a file containing several object files, that can be used as a single entity in a linking phase of a program.Normally the library is indexed, so it is easy to find symbols (functions, variables and so on) in them

There are two libraries

- Dynamic or shared
  - When the program is started, a program in the system (called a dynamic loader) checks out which shared libraries were linked with the program, loads them to memory, and attaches them to the copy of the program in memory.
  - C" library is normally a shared library, and is used by all C programs
- Static
  - collections of object files that are linked into the program during the linking phase of compilation, and are not relevant during runtime.
  - only the program's executable file is needed in order to run the program.

Creating static library

Using command called ar(archiver)

ar rc libutil.a util_file.o util_net.o util_math.o

Command to create a new static library called libutil.a with a flag r(replace if already exist) And c(create new one if the file name does not exist)

ranlib libutil.a

Command to create indexing in the archive(static library)

Cp -p

Command to copy static lib

Creating shared library

we need to use the compiler (either the compiler's driver, or its linker) to generate the library, and tell it that it should create a shared library, not a final program file.

Position Independent Code (PIC): PIC is a feature that ensures the generated code is not tied to a specific memory location. It achieves this by using relative addressing instead of absolute addressing. This is crucial for shared libraries, as they can be loaded into memory

at different addresses by various programs. The use of relative addresses allows the library to work regardless of its actual memory location.

```
gcc -fPIC -c util_file.c
gcc -fPIC -c util_net.c
gcc -fPIC -c util_math.c
gcc -shared(G)  libutil.so util_file.o util_net.o util_math.o
```

After you've defined LD_LIBRARY_PATH, you can check if the system locates the library properly for a given program linked with this library:

ldd prog

1 .In order to open and load the shared library, one should use the dlopen() function. It is used this way:

```
#include <dlfcn.h>      /* defines dlopen(), etc.       */
.
.
void* lib_handle;       /* handle of the opened library */

lib_handle = dlopen("/full/path/to/library", RTLD_LAZY);
if (!lib_handle) {
   fprintf(stderr, "Error during dlopen(): %s\n", dlerror());
   exit(1);
}
```